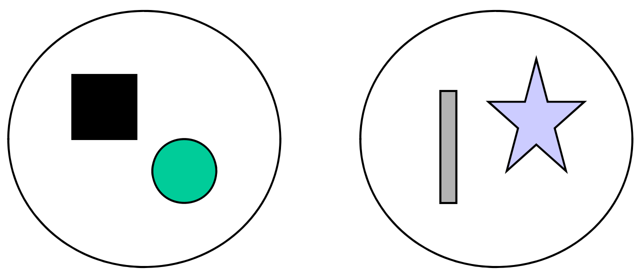
Disjoint Sets Data Structure

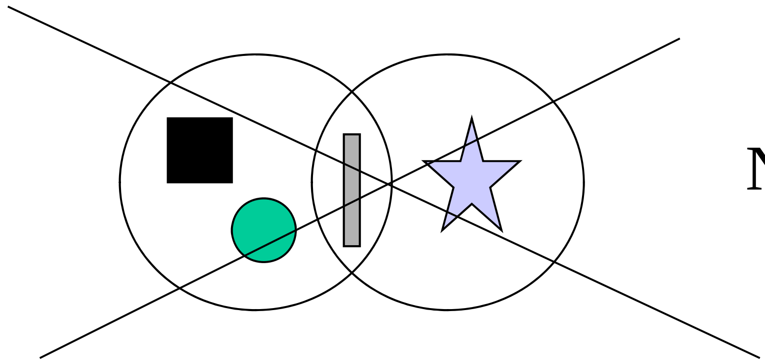
Bruce Nan

What Are Disjoint Sets?

Two sets A and B are disjoint if they have NO elements in common. $(A \cap B = 0)$



Disjoint Sets



NOT Disjoint Sets

What Are Disjoint Set Data Structures?

A disjoint-set data structure maintains a collection
S = {S1, S2,…,Sk} of disjoint dynamic (changing) sets.

- Each set has a representative (member of the set).
- Each element of a set is represented by an object (x).

Disjoint Sets

- Some applications require maintaining a collection of disjoint sets.

- A Disjoint set S is a collection of sets

  $S_1, \ldots\ldots S_n$ where $\forall_{i \neq j} S_i \cap S_j = \phi$

- Each set has a representative which is a member of the set (Usually the minimum if the elements are comparable)

Disjoint Set Operations

- Make-Set(x) – Creates a new set where x is it's only element (and therefore it is the representative of the set).

- Union(x,y) – Replaces S_x, S_y by $S_x \cup S_y$ one of the elements of $S_x \cup S_y$ becomes the representative of the new set.

- Find(x) – Returns the representative of the set containing x

Analyzing Operations

- We usually analyze a sequence of *m* operations, of which *n* of them are Make_Set operations, and *m* is the total of Make_Set, Find, and Union operations

- Each union operations decreases the number of sets in the data structure, so there can not be more than *n-1* Union operations

Applications

- Equivalence Relations (e.g Connected Components)

- Minimal Spanning Trees

Connected Components

- Given a graph G we first preprocess G to maintain a set of connected components.

  CONNECTED_COMPONENTS(G)

- Later a series of queries can be executed to check if two vertexes are part of the same connected component

  SAME_COMPONENT(U,V)

Connected Components

CONNECTED_COMPONENTS(G)

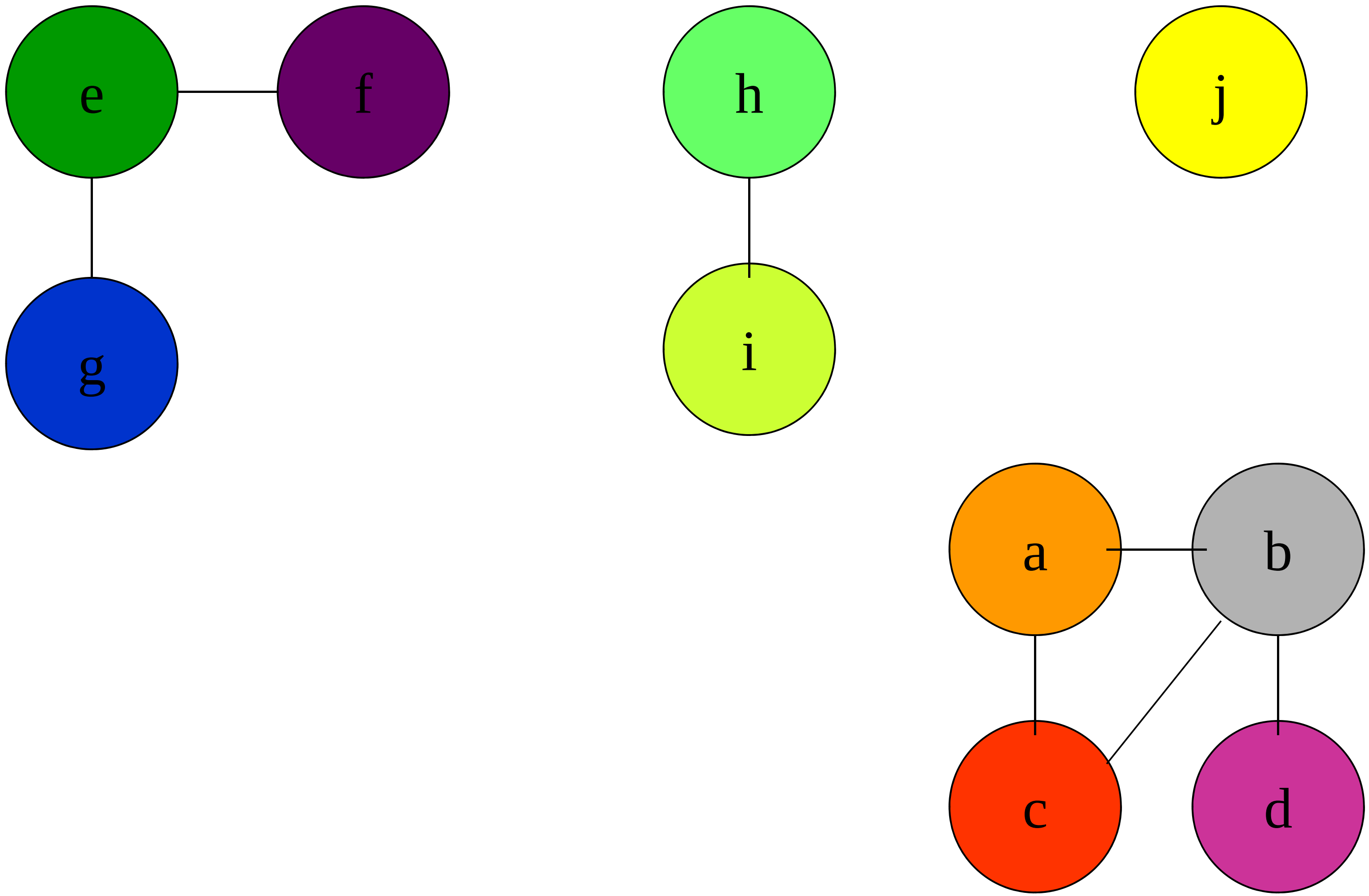
```
for each vertex v in V[G]
    do MAKE_SET (v)

for each edge (u,v) in E[G]
    do if FIND_SET(u) != FIND_SET(v)
        then UNION(u,v)
```

Connected Components

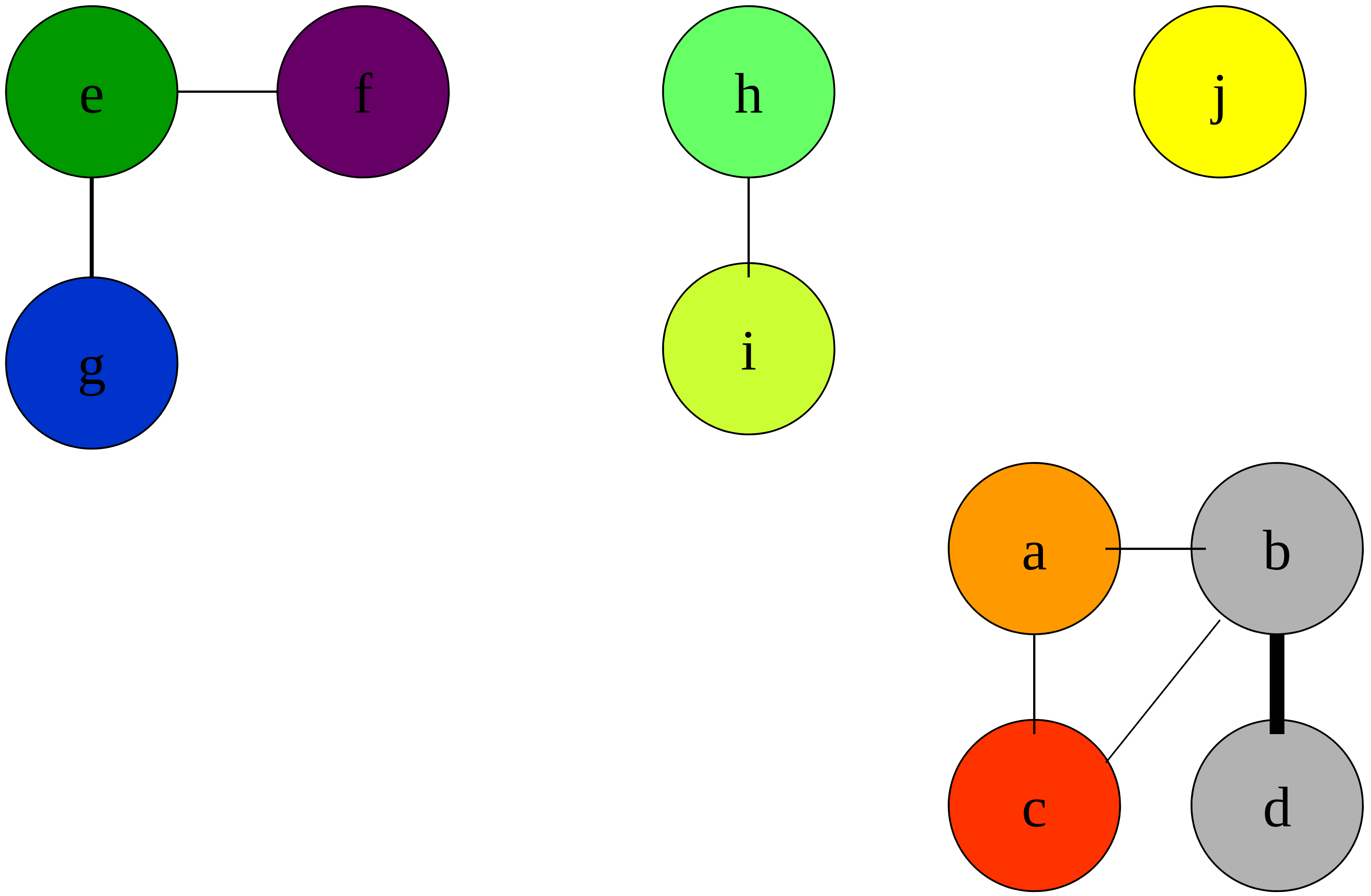
SAME_COMPONENT(u,v)

return FIND_SET(u) ==FIND_SET(v)

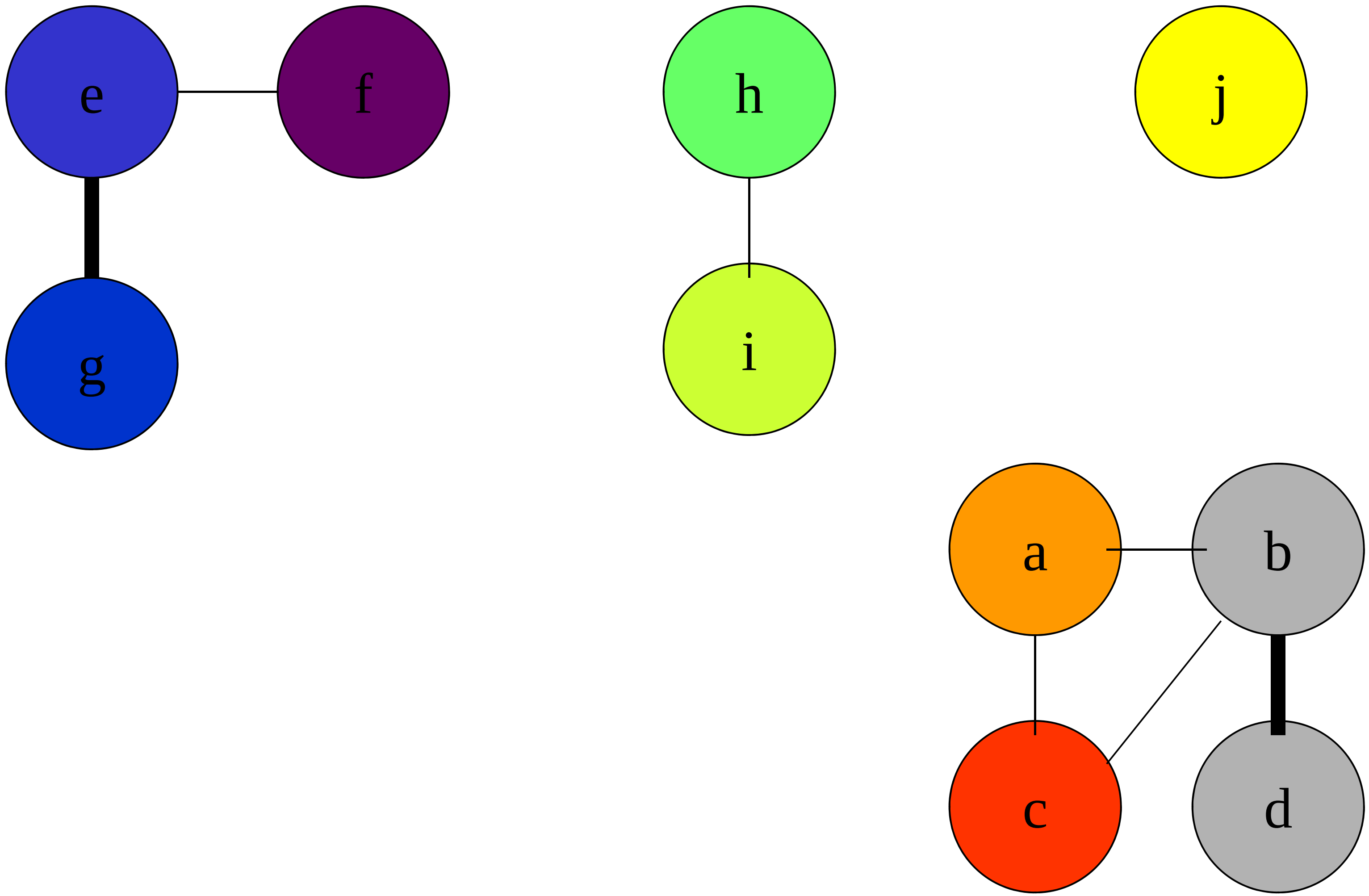
Example



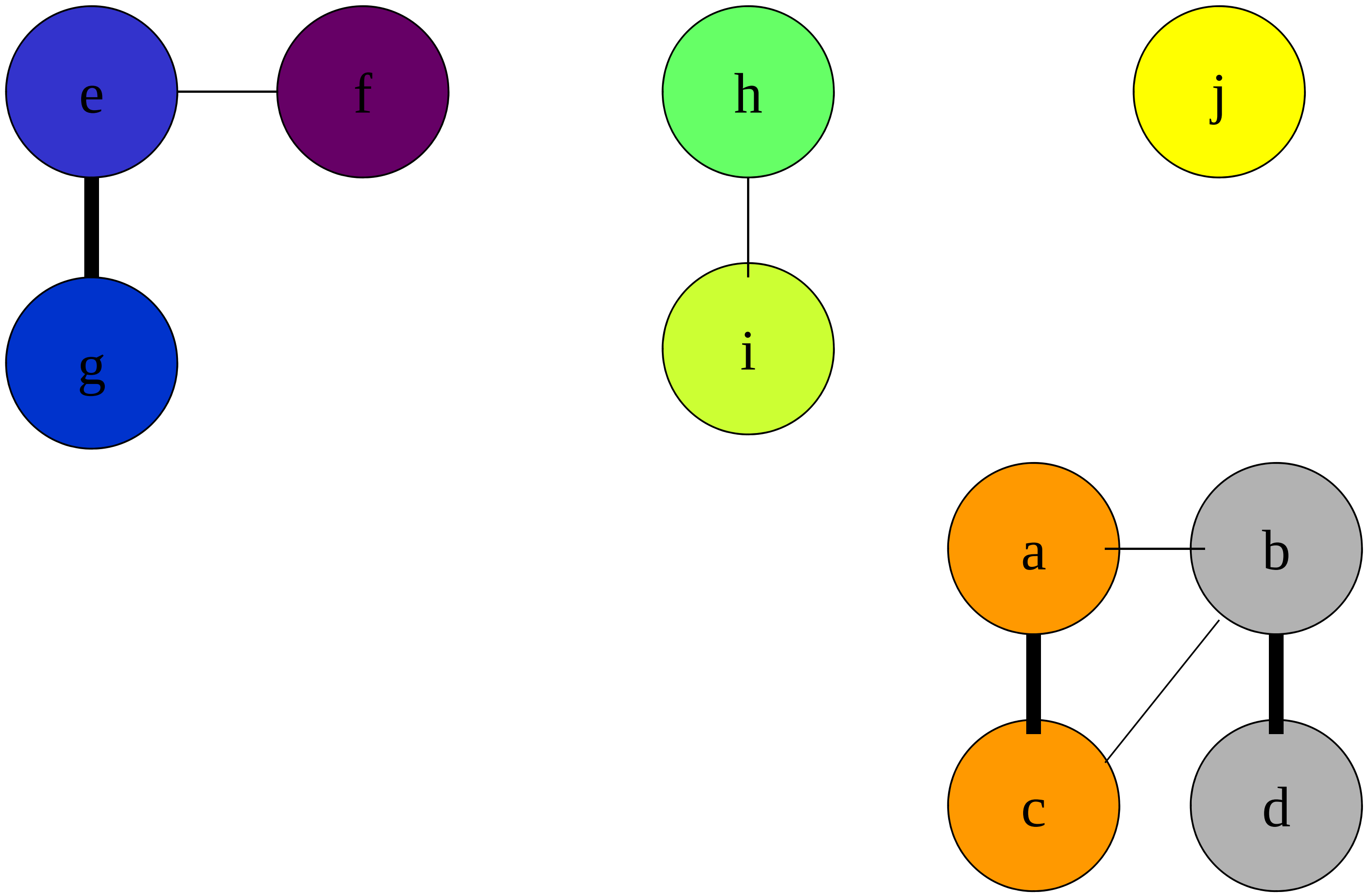
(b,d)



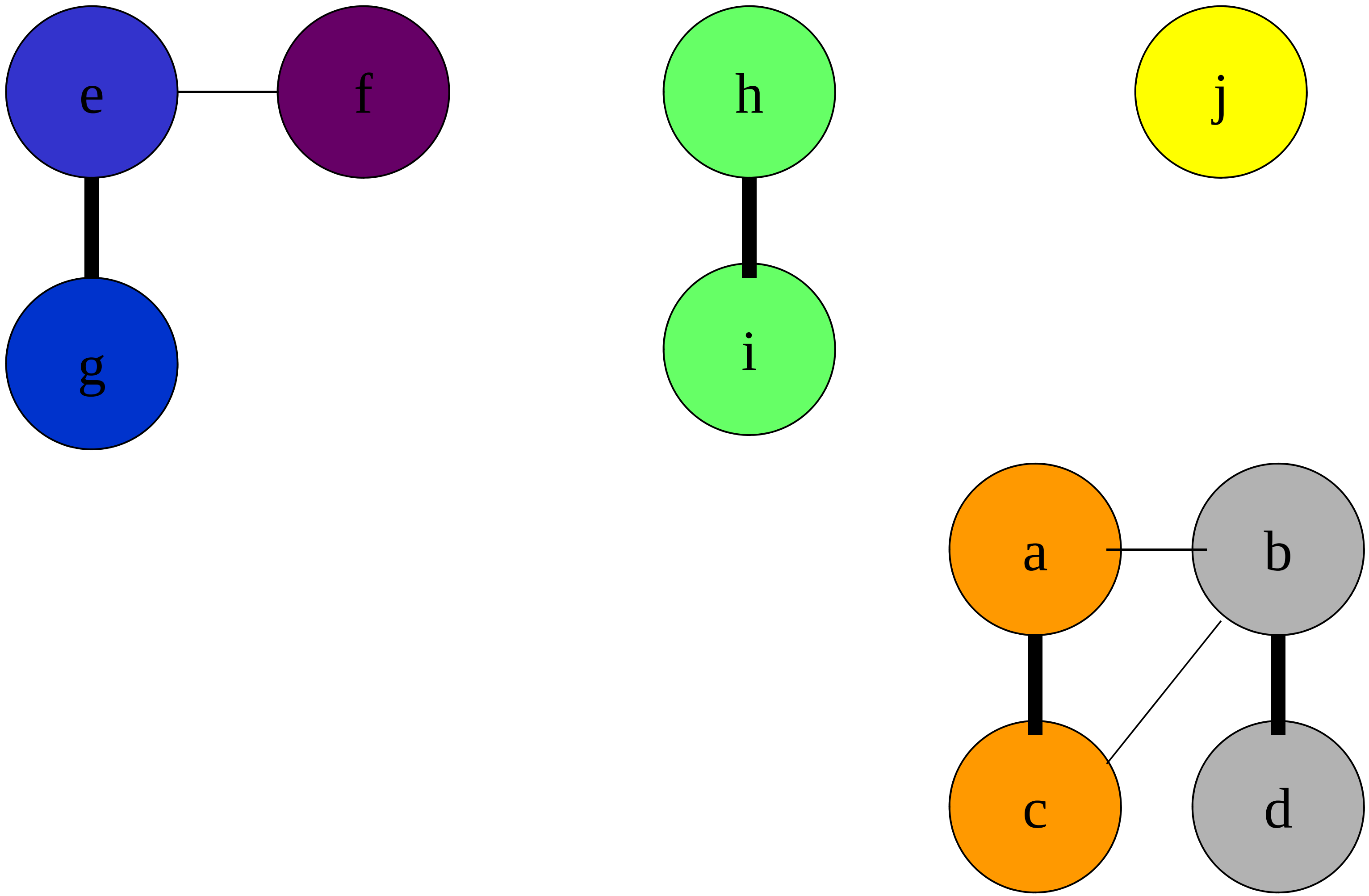
(e,g)



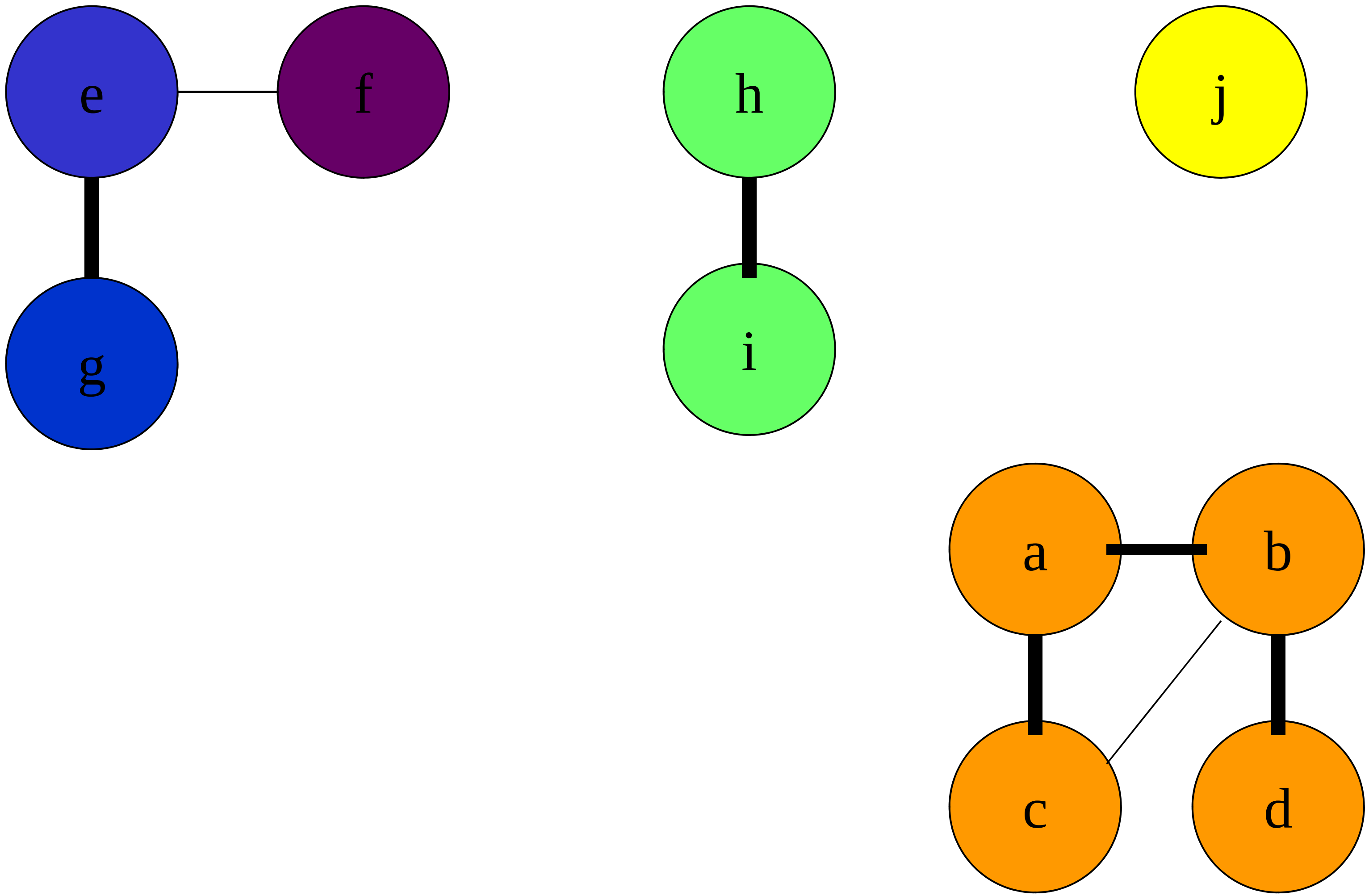
(a,c)



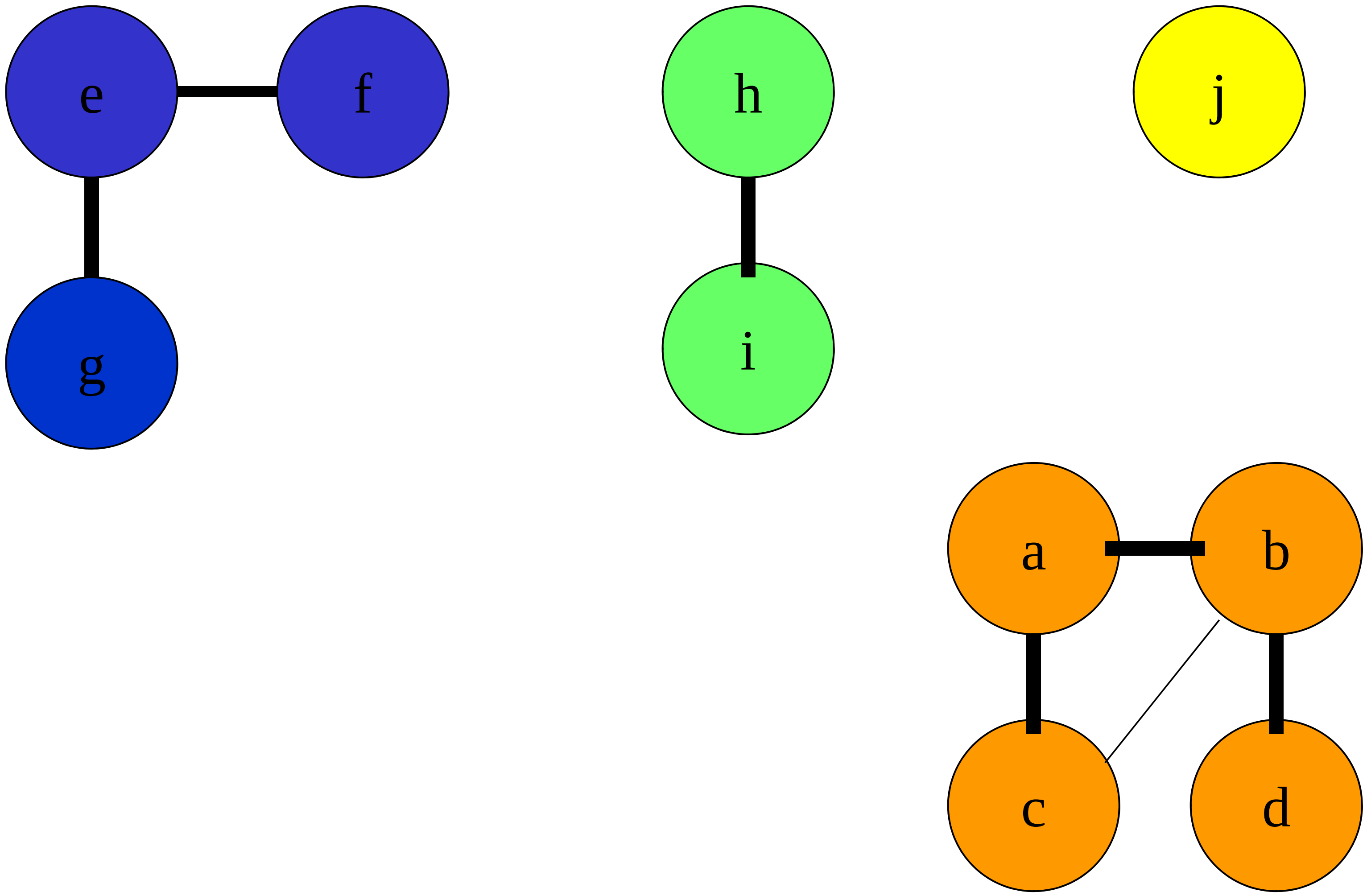
(h,i)



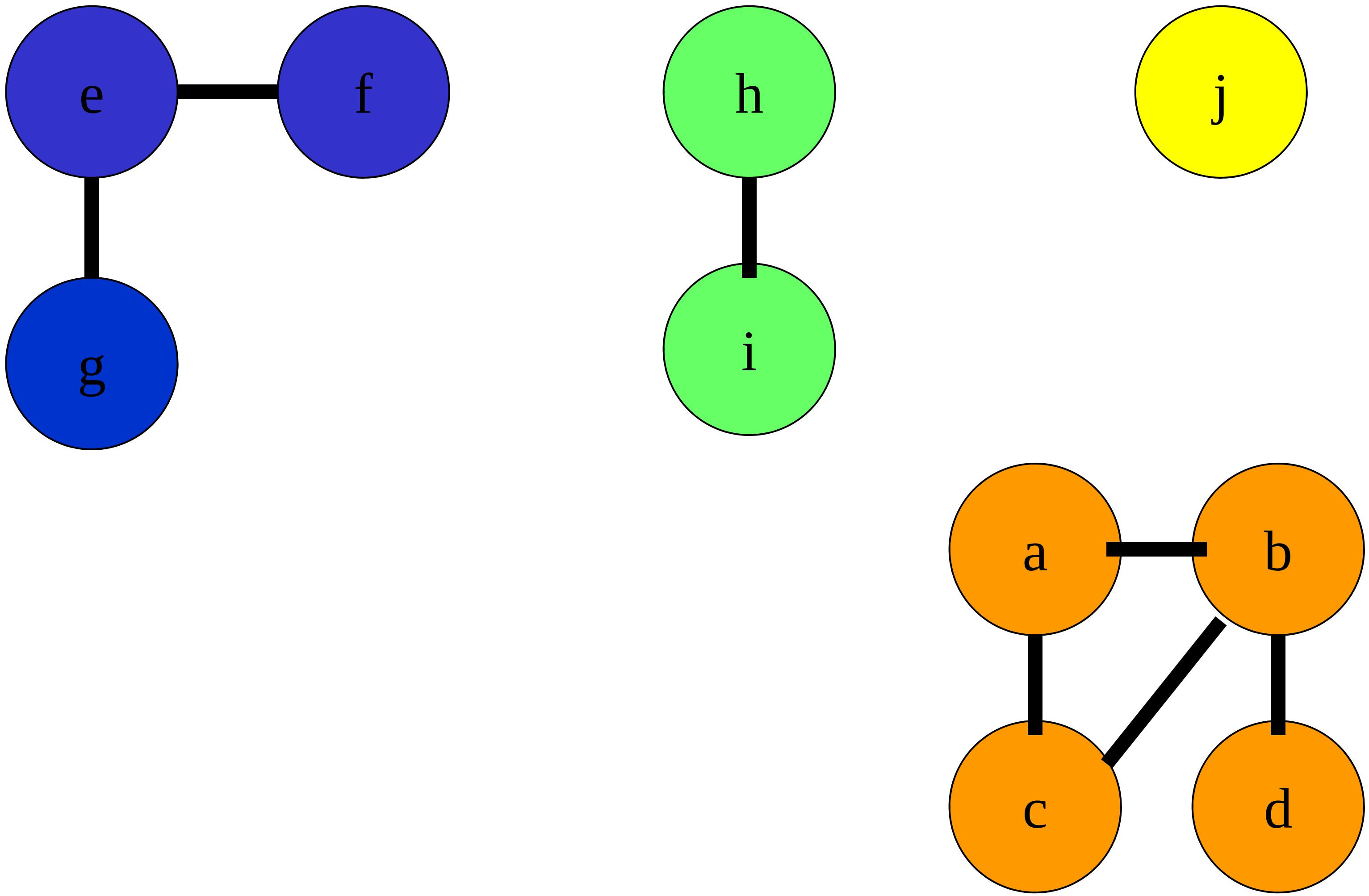
(a,b)



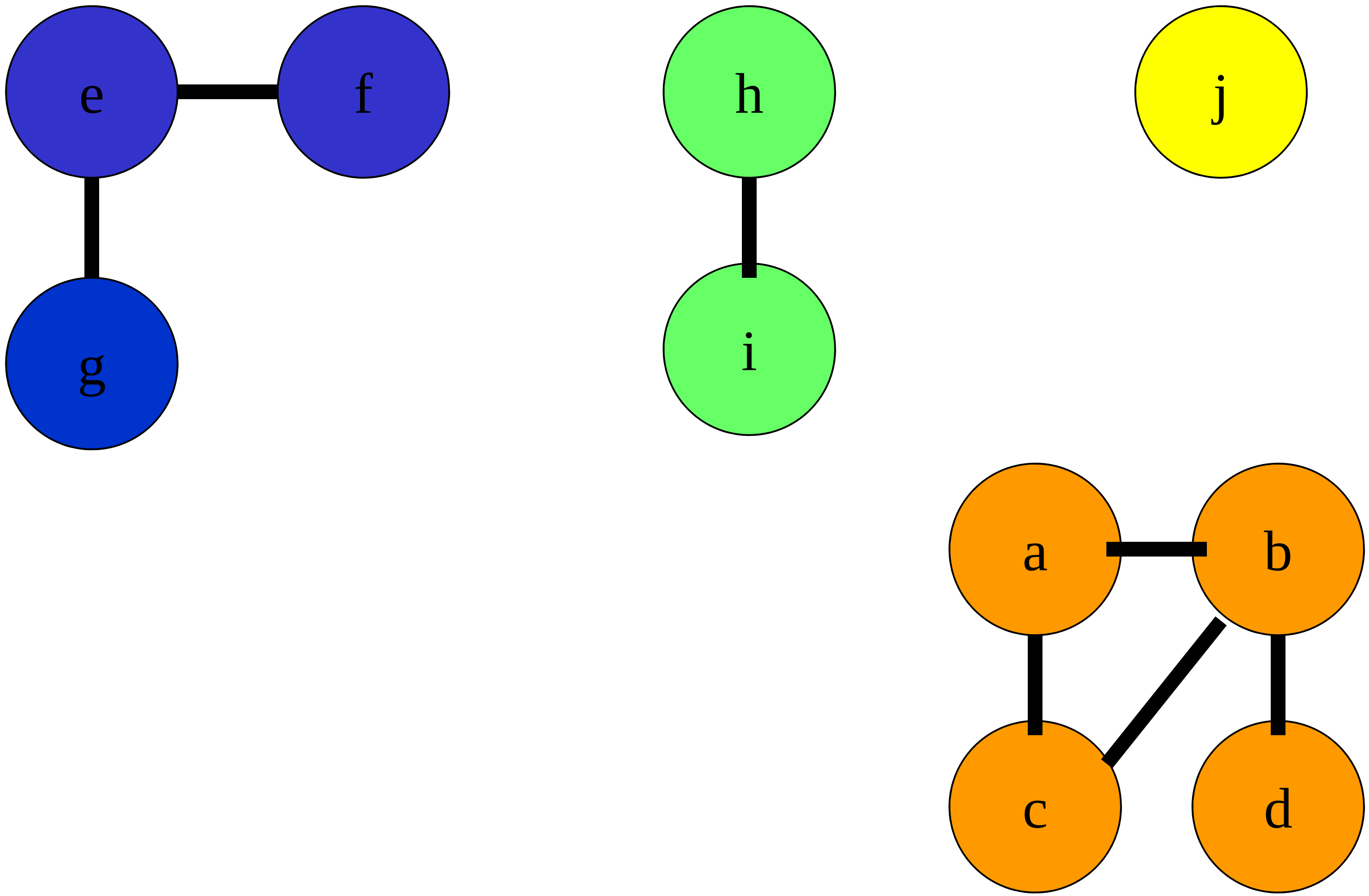
(e,f)



(b,c)



Result



Connected Components

- During the execution of CONNECTED-COMPONENTS on a undirected graph $G = (V, E)$ with k connected components, how many time is FIND-SET called? How many times is UNION called? Express you answers in terms of $|V|$, $|E|$, and k.

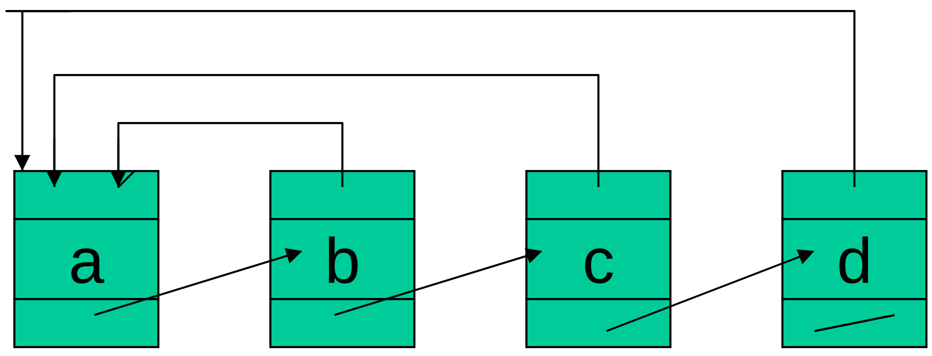
Solution

- FIND-SET is called 2|E| times. FIND-SET is called twice on line 4, which is executed once for each edge in E[G].

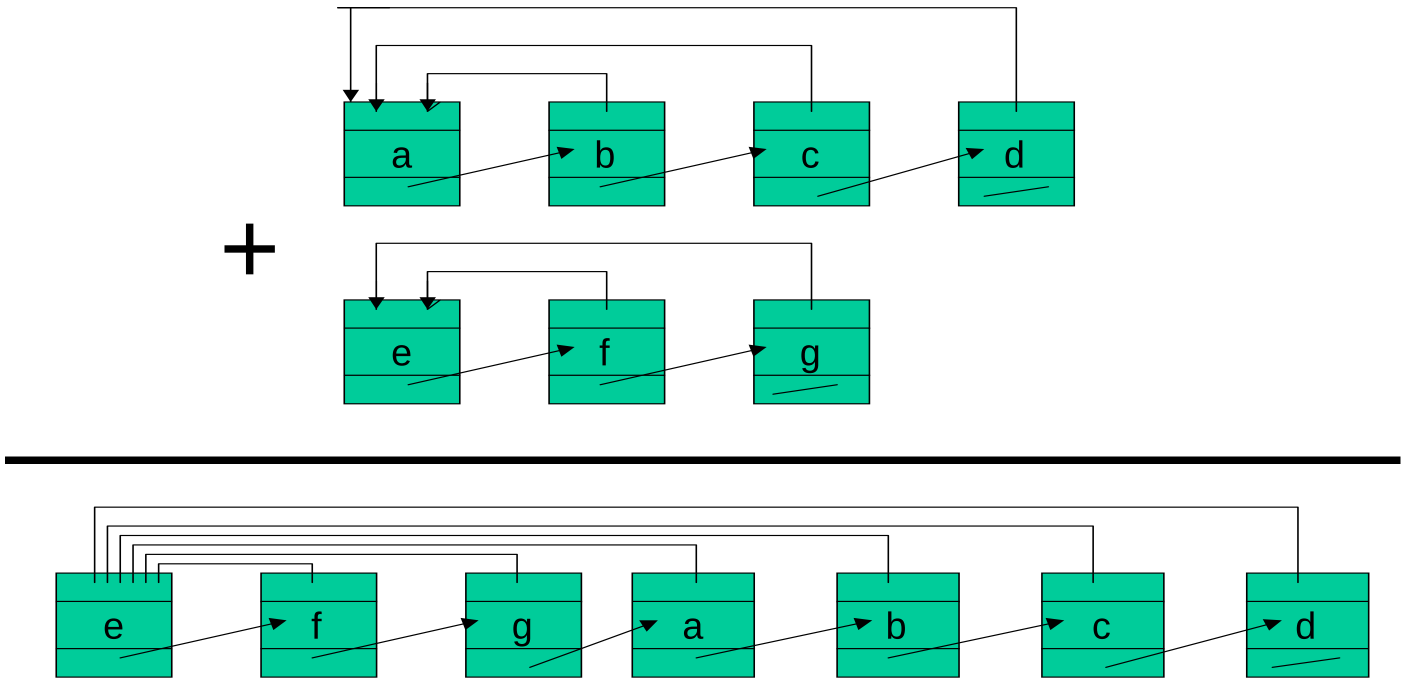
- UNION is called |V| - k times. Lines 1 and 2 create |V| disjoint sets. Each UNION operation decreases the number of disjoint sets by one. At the end there are k disjoint sets, so UNION is called |V| - k times.

Linked List implementation

- We maintain a set of linked list, each list corresponds to a single set.
- All elements of the set point to the first element which is the representative
- A pointer to the tail is maintained so elements are inserted at the end of the list



Union with linked lists



Analysis

- Using linked list, MAKE_SET and FIND_SET are constant operations, however UNION requires to update the representative for at least all the elements of one set, and therefore is linear in worst case time

- A series of m operations could take $\Theta(m^2)$

Improvement – Weighted Union

- Always append the shortest list to the longest list.
  A series of operations will now cost only $\Theta(m + n\log n)$

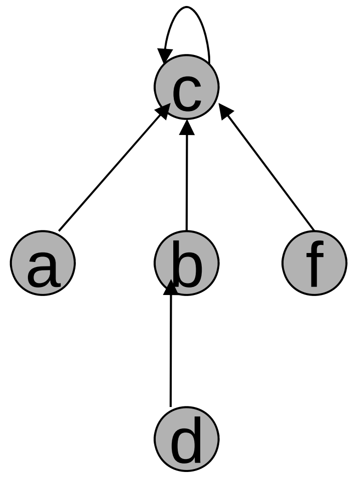
- MAKE_SET and FIND_SET are constant time and there are m operations.

- For Union, a set will not change it's representative more than log(n) times. So each element can be updated no more than log(n) time, resulting in nlogn for all union operations

Disjoint-Set Forests

- Maintain A collection of trees, each element points to it’s parent. The root of each tree is the representative of the set

- We use two strategies for improving running time
  – Union by Rank
  – Path Compression

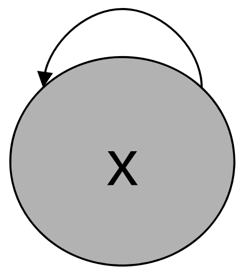


Make Set

- **MAKE_SET(x)**

  p(x)=x

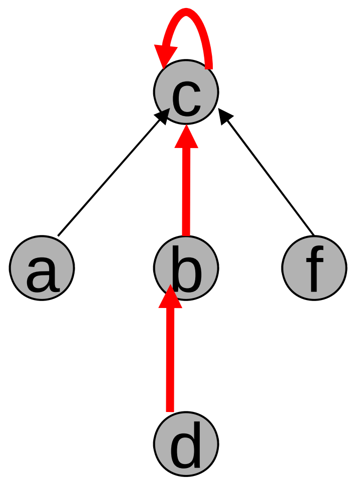
  rank(x)=0



Find Set

- **`FIND_SET(d)`**

```
if d != p[d]
  p[d]= FIND_SET(p[d])
return p[d]
```



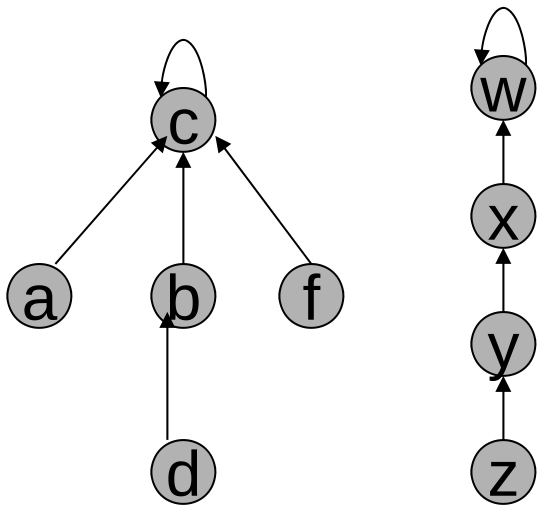
Union

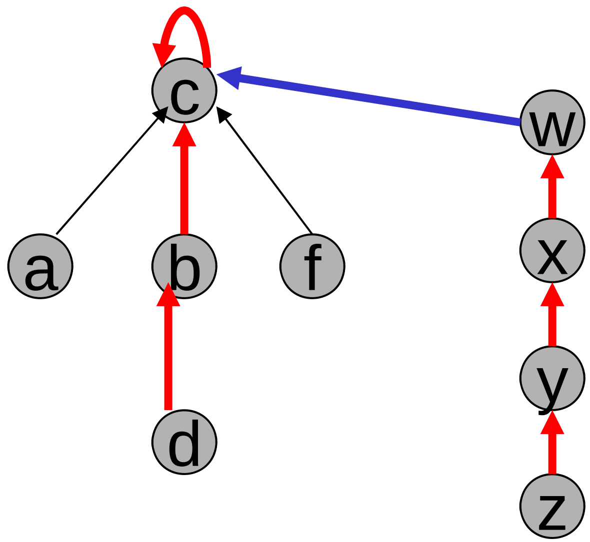
- **`UNION(x,y)`**

```
 link(findSet(x),
         findSet(y))
```

- **`link(x,y)`**

```
if rank(x)>rank(y)
 then p(y)=x
 else
  p(x)=y
  if rank(x)=rank(y)
   then rank(y)++
```





Analysis

- In Union we attach a smaller tree to the larger tree, results in logarithmic depth.
- Path compression can cause a very deep tree to become very shallow
- Combining both ideas gives us (**without proof**) a sequence of m operations in $O(m\alpha(m,n))$

Exercise

- Describe a data structure that supports the following operations:
  - find(x) – returns the representative of x
  - union(x,y) – unifies the groups of x and y
  - min(x) – returns the minimal element in the group of x

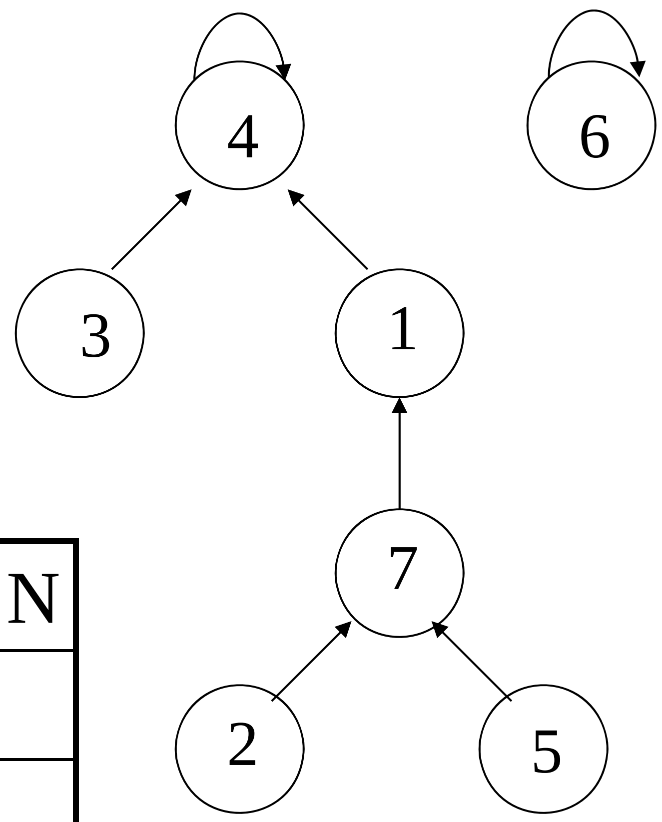
Solution

- We modify the disjoint set data structure so that we keep a reference to the minimal element in the group representative.
- The find operation does not change (log(n))
- The union operation is similar to the original union operation, and the minimal element is the smallest between the minimal of the two groups

Example

- Executing find(5)

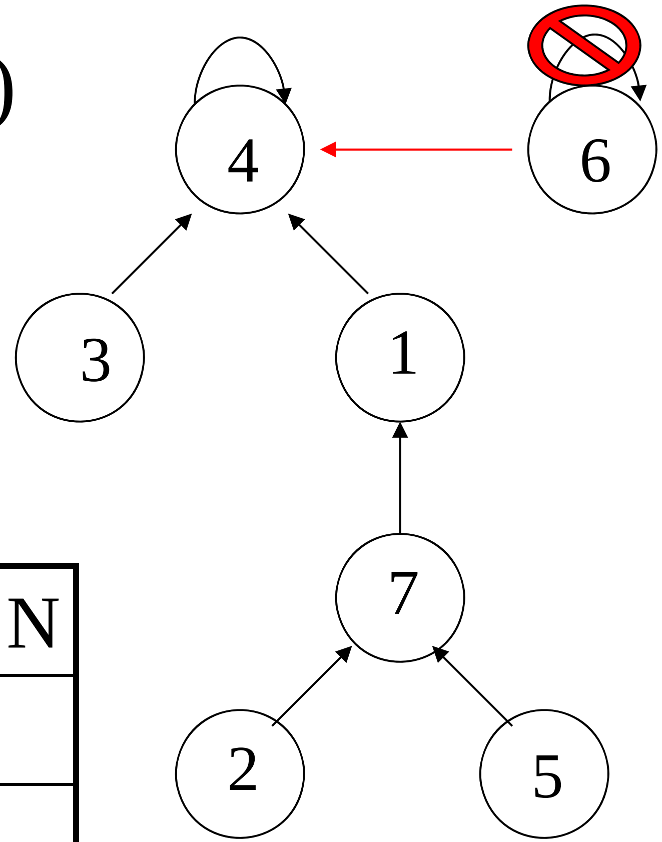
  7 → 1 → 4 → 4



	1	2	3	4	5	6	..	N
Parent	4	7	4	4	7	6		
min				1		6		

Example

- Executing union(4,6)



	1	2	3	4	5	6	..	N
Parent	4	7	4	4	7	4		
min				1		1		

Exercise

- Describe a data structure that supports the following operations:
  - find(x) – returns the representative of x
  - union(x,y) – unifies the groups of x and y
  - deUnion() – undo the last union operation

Solution

- We modify the disjoint set data structure by adding a stack, that keeps the pairs of representatives that were last merged in the union operations

- The find operations stays the same, but we can not use path compression since we don’t want to change the modify the structure after union operations

Solution

- The union operation is a regular operation and involves an addition push (x,y) to the stack

- The deUnion operation is as follows
  - (x,y) ← s.pop()
  - parent(x) ←x
  - parent(y) ←y

Example

- Example why we can not use path compression.

  – Union (8,4)

  – Find(2)

  – Find(6)

  – DeUnion()

	1	2	3	4	5	6	7	8	9	10	
parent	4	7	7	4	8	1	5	8	1	4	